Hybrid Message Pessimistic Logging. Improving Current Pessimistic Message Logging Protocols.

Hugo Meyer[a,b,*], Ronal Muresano[b], Marcela Castro-León[b], Dolores Rexachs[b], Emilio Luque[b]

[a] *Barcelona Supercomputing Center (BSC), Barcelona, Spain*
[b] *Computer Architecture and Operating Systems Department, University Autonoma of Barcelona, Barcelona, Spain*

Abstract

With the growing scale of HPC applications, there has been an increase in the number of interruptions as a consequence of hardware failures. The remarkable decrease of Mean Time Between Failures (MTBF) in current systems encourages the research of suitable fault tolerance solutions. Message logging combined with uncoordinated checkpoint compose a scalable rollback-recovery solution. However, message logging techniques are usually responsible for most of the overhead during failure-free executions. Taking this into consideration, this paper proposes the Hybrid Message Pessimistic Logging (HM_{PL}) which focuses on combining the fast recovery feature of pessimistic receiver-based message logging with the low failure-free overhead introduced by pessimistic sender-based message logging. The HM_{PL} manages messages using a distributed controller and storage to avoid harming system's scalability. Experiments show that the HM_{PL} is able to reduce overhead by 34% during failure-free executions and 20% in faulty executions when compared with a pessimistic receiver-based message logging.

Keywords: Fault Tolerance, Availability, Scalability, Performance, MPI, Message Logging.

*Corresponding author

Email addresses: hugo.meyer@bsc.es, hugo.meyer@caos.uab.es (Hugo Meyer), ronal.muresano@caos.uab.es (Ronal Muresano), marcela.castro@uab.es (Marcela Castro-León), dolores.rexachs@uab.es (Dolores Rexachs), emilio.luque@uab.es (Emilio Luque)

1. Introduction

Current HPC machines gather a large number of processors with the objective of reducing execution time (scalability) or managing more data when solving complex problems. However, with the increase in the number of components, the miniaturization and high concentration of hardware components, the number of failures that affect systems also increases [1] [2].

In [3] an analysis of 22 HPC systems is presented, where it can be clearly seen that failure rates in these systems increase as the number of nodes and processors increases. It has been determined from the data collected that the main reason for stoppages in these systems are the hardware failures.

Therefore, fault tolerance mechanisms are a valuable feature to guarantee the successful completion of parallel applications in HPC clusters. The need for reliable fault tolerant HPC systems has intensified because a hardware failure may result in a possible increase in execution time because of re-executions, lower throughput and higher economic costs [3].

In order to handle hardware failures in parallel environments (where intrinsic redundancy exists) reducing loss of computational work, rollback-recovery protocols [4] [3] are used. Many rollback-recovery protocols that protect parallel applications are based on checkpointing and message logging or a combination of the two.

Coordinated checkpoint is one of the most used and implemented rollback-recovery protocol in HPC systems. It consists of interrupting the whole parallel application at some points to save a global snapshot of the application, guaranteeing that there are no message transmissions in course. However, regarding scalability, this approach has some drawbacks such as: the coordination time increases with the number of processes; in the event of failure, even non-failed processes should rollback, affecting resource usage and energy consumption; and, checkpoint-restart suffer from high I/O overhead.

The current trend in HPC is to avoid fault tolerance solutions where collective operations (such as the coordination) and centralized components are used because they could compromise the scalability of applications. On the other hand, uncoordinated checkpoint protocols allow each process to save its state independently, without needing coordination. Nevertheless, in order to avoid the domino effect [4], these protocols should be used in combination with a message logging approach.

Uncoordinated approaches guarantee that scalability is not compromised, since each process may take actions on its own, avoiding the costly step

of process coordination. Moreover, only faulty processes must rollback to a previous state, reducing the amount of loss computation that has to be re-executed, and possibly permitting overlap between recovery and regular application progress. In order to avoid the rollback of non-failed processes, uncoordinated approaches should be combined with event logging. Thus, in the event of failure, restarted processes could use the logged messages without forcing other non-failed processes to rollback for recreating old messages.

Message logging protocols are based on the assumption that process states can be reconstructed replaying all messages in the correct order [5]. Non-deterministic events (reception of messages, i/o operations, etc.) should be replayed in the same way as they occur before the failure. Message logging techniques consider parallel applications as a sequence of deterministic events (computation) separated by non-deterministic events (messages) [6].

According to [7], message logging is expected to be more appropriate than coordinated checkpoints when the MTBF is greater than 9 hours. Moreover, with smaller MTBF message logging approaches present more advantages than coordinated checkpoints. For example, while coordinated checkpoints stop progressing when the MTBF is shorter than 3 hours, message logging techniques allow applications to continue progressing.

Uncoordinated fault tolerance protocols, such as message logging, seem to be the best option for failure prone environments, since coordinated protocols present a costly recovery procedure which involves the rollback of all processes. Pessimistic log protocols ensure that all messages received by a process are first logged before it causally influences the rest of the system.

Taking into account that most of the overhead during failure-free executions is caused by message logging approaches, in this work, we discuss the most used message logging techniques to provide fault tolerance support. We also present our proposed message logging approach, which focuses on combining the advantages of pessimistic receiver and sender based approaches, this technique is called Hybrid Message Pessimistic Logging (HM_{PL}). In [8] we have presented an introduction and details about this technique, and in this paper we have extended the analysis made to a wider set of applications and scenarios, such as failure-free scenarios as well as faulty scenarios. Furthermore, in this paper the recovery model is fully described.

When designing the HM_{PL} we set the next requirements:

1. Pessimistic: this ensures that in case of a failure, there is no need to rollback non-failed processes. It also simplifies garbage collection.

2. Distributed: the logging processes are distributed among the parallel environment, then the bottlenecks are avoided.

3. Decentralized: messages are saved in senders and receivers of the processes involved in the message transmission.

4. Transparent: messages are intercepted by a lower fault tolerant layer, avoiding changes in application code.

5. Automatic: the protection and recovery mechanisms take place without any user intervention.

We focus on pessimistic versions because they ensure that in case of a failure, there is no need to rollback non-failed processes.

The Hybrid Message Pessimistic Logging focuses on combining the fast recovery feature of pessimistic Receiver-Based Message Logging (RBML) with the low protection overhead introduced by pessimistic Sender-Based Message Logging (SBML). Furthermore, the main objective of the HM_{PL} is to perform better than receiver-based approaches during failure-free executions but also to avoid a high penalization in case of failure.

In this paper we will present and validate the protection and recovery mechanisms of the HM_{PL} by integrating it inside the Redundant Array of Distributed and Independent Controllers (RADIC) fault tolerant architecture [9]. Experiments show that the HM_{PL} is able to reduce overhead by 34% during failure-free executions and 20% in faulty executions when comparing it with a pessimistic receiver-based message logging.

The main contributions of this work are detailed below:

- The HM_{PL} focuses on providing fast recovery with low failure-free overhead by combining characteristics of pessimistic sender-based and receiver-based message logging approaches. The HM_{PL} aims to balance fast and local recovery of failed processes (introducing minimal interference in the rest of the system) with low failure-free overhead.

- We propose the usage of temporal buffers in senders in order to reduce the penalties of classic pessimistic receiver-based logging in the critical path of application executions.

- Functional and analytical comparisons between the HM_{PL} and classic message logging techniques are presented in this paper.

The main proposal of this work is a message logging protocol that combines advantages of pessimistic sender-based and pessimistic receiver-based logging in order to reduce failure-free execution time and reduce recovery complexity.

The rest of the paper is structured as follows: the next section presents the related work. In Section 3 some classic message logging techniques used as backbone of our proposal are described. In Section 4, the $HM_{PL}$is described in detail. In Section 5 the experimental validation is presented considering both non-faulty and faulty environments. Finally, in Section 6, we present our conclusions and future work.

2. Related Work

In the recent past, coordinated approaches were preferred when providing fault tolerance support to parallel applications because of their ease of deployment and low complexity. However, coordinated checkpoint approaches present three major disadvantages:

- When using fault tolerance support below the application level (application-transparent fault tolerance), increasing the number of processes involved may increase the time needed to coordinate all the processes. Normally, most protocols require at least two global coordinations [4].
- In the event of failure, all processes of the parallel application must rollback to a previous state, even the non-failed processes. Then, there may be a considerable computational time loss and inefficient usage of resources.
- Checkpoint and restart both suffer from high I/O overhead at the scale envisioned for future systems, leading to poor overall efficiency barely competing with replication [10].

The current trend in HPC is to avoid fault tolerance solutions where collective operations (such as the coordination) and centralized components are used because they could compromise the scalability of applications. On the other hand, uncoordinated checkpoint protocols allow each process to save its state independently without needing coordination. Nevertheless, when combining these protocols with message logging, the overheads in failure-free executions become relevant.

Several works have been developed to improve the performance and minimize the overhead of message logging protocols. In [11] a sender-based logging approach to avoid sympathetic rollback of non-failed processes was presented. Sympathetic rollback means that a process rolls back to re-send a message that has been lost because the receiver has rolled back to a previous state. This paper does not address the problem of rolling back because of orphan message creations. They propose to detect messages that can never cross a possible recovery line.

In [12] an extensive analysis of message logging protocols is presented and a comparison of sender-based, receiver-based and causal protocols is performed. According to the results, they conclude that relying on other processes to provide messages to restarted processes is not the best option. Sender-based and causal protocols incur in high recovery costs because of this. They also recommend that optimistic protocols should focus on developing orphan-detection protocols, instead of only focusing on performance metrics.

In [7] a coordinated checkpoint protocol is compared with a pessimistic sender-based message logging implemented in MPICH-V [13]. The obtained results demonstrate that message logging approaches can sustain a more adverse failure pattern than coordinated checkpoints.

In [14] a comparison between pessimistic and optimistic sender-based message logging approaches is presented, and both seem to have a comparable performance. Nevertheless, when a failure occurs using sender-based protocols, processes that were not involved in the failure may need to re-send messages to restarted processes. Then, the latency and complexity in the recovery phase are increased.

In [15], they propose to reduce the failure-free overheads in pessimistic sender-based approaches by removing useless memory copies and reducing the number of logged events. Nevertheless, the latency and complexity in the recovery phase are increased because of the usage of sender-based protocols.

In [16] a mechanism to reduce the overhead added using the pessimistic receiver-based message logging implemented in RADIC was proposed. The technique consists of dividing messages into smaller pieces, so receptors can overlap receiving pieces with the message logging mechanism. This technique and all the RADIC Architecture has been introduced into Open MPI in order to support message passing applications. RADIC architecture is explained more in detail in subsection 4.4.

In [12], besides the extensive analysis of classic message logging protocols, Rao et al. also describe hybrid message logging protocols (Sender-based

and Causal). They propose an orphan-free protocol that tries to maintain performance of sender-based protocols while approaching the performance of receiver-based protocols. In this case the sender syncrhronously saves messages and the receiver asynchronously saves them into stable storage. Details about the design of the hybrid logging that they propose are missing. They do not cover the mechanisms to avoid orphan processes creation, nor do they specify how and where the received messages are stored. On the other hand, the HM_{PL} is based on a distributed stable storage where each process saves messages on a different node and orphan processes are avoided by detecting the source of non-determinism while receiving messages. Moreover, the HM_{PL} focuses on providing a scalable message logging solution, and by being a pessimistic technique it avoids the rollback of non-failed processes.

Classical pessimistic message logging protocols propose saving messages in a synchronous manner during failure-free executions, thus introducing high overheads in messages that are transferred between processes inside the same multicore node. Depending on where the messages should be logged (e.g. a different node or a centralized stable storage), the latency of an internode message may be added to the latency of each intercore message.

Coordinated and centralized checkpoints avoid the overhead of message logging techniques. However, the coordination overheads and checkpoint saving time may increase considerably when the number of processes increases.

Works like [17], [18] and [19] focus on grouping the processes that communicate more frequently in order to reduce the number of messages logged using a coordinated checkpoint between these processes.

Taking into account that in parallel systems the most common unit of failure is a node, works such as [20] and [21] focus on developing techniques to provide hybrid or semi-coordinated checkpoint approaches, where a coordinated checkpoint is used inside the node and messages between nodes are logged to stable storage. The objective here is to avoid logging intra-node messages, thus lowering the message logging overhead.

3. Classic Message Logging Techniques

In this section, we describe in detail the pessimistic version of the RBML and the SBML. We focus on pessimistic versions because they ensure that in case of a failure, there is no need to rollback non-failed processes. These two protocols have been used as the backbone of our proposed HM_{PL}. In

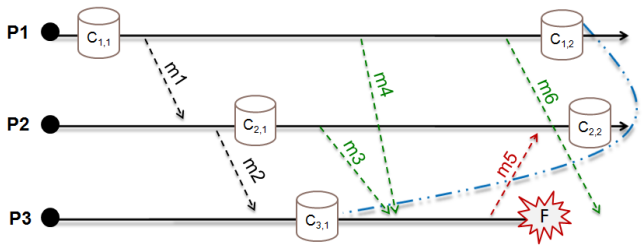


Figure 1: Message Types in uncoordinated checkpoint protocols.

the next paragraphs we present an analysis of the pessimistic versions of the RBML and the SBML in order to explore their main characteristics.

3.1. Messages Types

When using message logging protocols, we should be able to deal with different types of messages that could appear during the applications' execution. Let us consider the recovery line composed by $C_{1,2}$, $C_{2,2}$ and $C_{3,1}$ while using uncoordinated checkpoints in Figure 1. Below we clarify some concepts that are used in message logging protocols [21] [18]:

- **In-transit Messages**: Messages m3 and m4 are crossing the recovery line and they are considered in-transit messages. Supposing that P3 is affected by a failure, and it restarts from $C_{3,1}$, messages m3 and m4 will not be available anymore since P2 and P1 have already sent them in the past. In order to build a consistent recovery line, m3 and m4 should be saved as well as the checkpoints.

- **Lost Messages**: Messages m3 and m4 from Figure 1 could become lost messages during a re-execution since the send events are recorded but not the reception.

- **Delayed Messages**: These are messages that are sent by non-failed processes to failed processes that will arrive after the failed processes restart and reach the corresponding reception event. In Figure 1, m6 is a delayed message.

- **Duplicated Messages**: These kinds of messages are produced when a process rolls back and re-sends previously sent messages. Considering Figure 1, if P3 fails, it will roll back to $C_{3,1}$ and send m5 to P2 again. As m5 was already received by P2, this will be a duplicated message and it should be discarded.

- **Orphan Messages**: Considering the recovery line (Figure 1), message m5 is crossing it from a possible future of P3 to the past of process P2. Message m5 depends on messages m3 and m4 (taking into account Lamport's happened-before relationship [6]) and in a future re-execution, the order of reception of these messages could vary (in MPI could be caused by the usage of a wild-card such as MPI_ANY_SOURCE). If this happens, instead of producing message m5, P3 could produce a different message (m5') and P2 will become an orphan process because it depends on a message that does not exist anymore.

Message logging techniques focus on providing a consistent recovery set from checkpoints taken at independent moments during the execution and they should be able to deal with the type of messages explained above.

Message logging is useful when the interactions with the outside world are frequent, because it allows a process to repeat its execution and be consistent without having to take expensive checkpoints before sending such messages [4]. Generally, log-based recovery is not susceptible to the domino effect, making them a good complement to uncoordinated checkpoint techniques.

3.2. Classification of Message Logging Protocols

When using uncoordinated checkpoint protocols, all in-transit messages should be saved to avoid sender rollback, and non-deterministic events should be tracked to avoid the creation of orphan messages. Duplicated messages should also be discarded.

According to the moment in which messages are saved and operation mode, message logging protocols can be classified in the next three types [4]:

- **Pessimistic**: In this kind of protocol the assumption is that the failure could occur immediately after the occurrence of a non-deterministic event, thus no process can depend on that event until it is correctly saved in stable storage. Each event is saved in a synchronous manner, thus increasing the overheads during failure-free execution but avoiding the generation of orphan messages. Pessimistic protocols guarantee that only failed processes rollback to their last checkpoint, facilitating garbage collection and enabling fast recovery.

- **Optimistic**: Each non-deterministic event is saved in an asynchronous way, assuming that the events will be saved correctly before the occurrence of a failure. This kind of protocols reduces the overhead during failure-free executions, but orphan processes can be created.

- **Causal**: These protocols avoid the synchronous behavior of pessimistic approaches, but additional copies of messages are generated to avoid orphan messages. These approaches try to combine the advantages of the previous ones, but they maintain the drawbacks of optimistic protocols, since a more complex recovery algorithm is needed.

The mentioned message log protocols can be receiver-based or sender-based, depending on who saves the message. Receiver-based logging protocols can double the message delivery latency during failure-free executions because every received message should be re-sent to a stable storage, however the Mean Time to Recover (MTTR) of a process tend to be lower than in sender-based approaches. Sender-based logging protocols introduce less overhead during failure free-executions but during recovery, sender processes should replay all messages exchanged with restarted processes [4]. Garbage collection in sender-based protocols is more complex since extra communications between senders and receivers is needed to erase non-necessary messages. By contrast, for receiver-based message logging, all necessary data to rebuild the state of a restarted process is available in its log repository, therefore this protocol is less complex and normally faster during recovery. Garbage collection in receiver-based approaches is less complex since after each checkpoint, receivers can delete their message logs.

3.3. Pessimistic Sender-Based Message Logging

The Pessimistic SBML [22] is a solution that focuses on introducing low overhead during failure-free executions. Non-deterministic events are logged in the volatile memory of the machine from which the message is going to be sent. The main idea behind this message logging technique is to avoid the introduction of high overheads in communications and delays in computations by asynchronously writing the messages in stable storage.

In Figure 2 we illustrate the operation of a pessimistic version of the SBML with the main data structures that it requires. We assume that the message M includes the headers with information about sender, destination and also the payload of the message. The logical times (t_x) that can be observed on the left of the figure are there only to indicate precedence of steps. We have split each process task into Application tasks (App) and fault tolerance (FT) tasks.

The main data structures and values used in the SBML are:

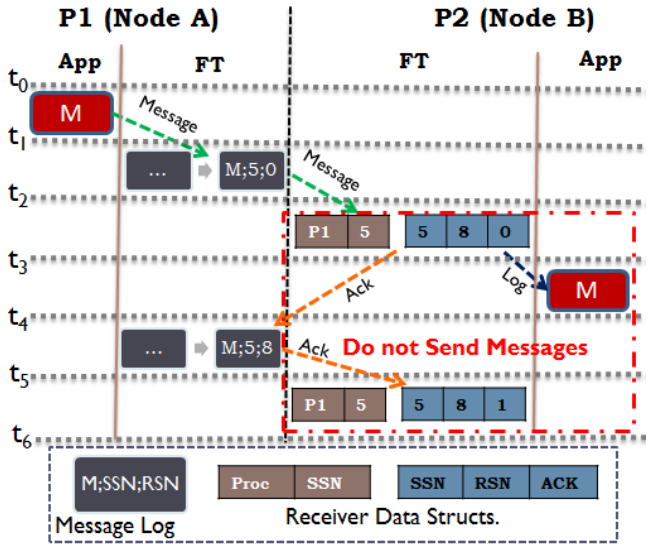


Figure 2: Sender-Based Message Logging during failure-free execution.

- *Send Sequence Number (SSN)*: this value indicates the number of messages sent by a process. This value is used for duplicated message suppression during the recovery phase. When a process fails and recovers from a checkpoint, it will re-send some messages to some receivers. If the receiver processes have the current SSN value for that sender, they will be able to discard these duplicated messages.

- *Receive Sequence Number (RSN)*: this value indicates the number of messages received by a process. The RSN is incremented with each message reception and then this value is appended to the ACK and sent back to the sender.

- *Message Log*: where the sender saves each outgoing message. Along with the payload of the message also the identification of the destination process and the SSN used for that message are saved. When the RSN of the message is returned by the receiver, it is also added to the log.

- *Table of SSN*: this table is located in the receivers and it registers the highest SSN value received for messages of each process. The information saved in this table is used for duplicated message detection.

- *Table of RSN*: this table is located in the receivers and has one entry for each received message. It is indexed by SSN and contains the RSN

and a value that indicates if the ACK of the message has been received by the sender.

When a process is checkpointed, all these data structures are also saved, except the *Table of RSN*, since all received messages are now part of the checkpoint. All messages sent to an already checkpointed process should be deleted from the message log.

In Figure 2, when the message M is about to be sent (from P1 to P2), P1 first saves the message M and the SSN.

Once the message is saved into a buffer of the sender, the message M and the SSN are sent to P2. P2 saves the sender ID (P1) and the SSN in the *Table of SSN*, as well as incrementing its RSN. After this, P2 adds this RSN to the *Table of RSN* and it sends and ACK with the current RSN to P1. P1 receives and adds the RSN to the *Message Log*, then P1 sends an ACK that P2 receives and adds to the *Table of RSN*.

There is almost no delay in computations while the message logging is taking place since P1 can continue its execution after saving the message and P2 after receiving it. However, between the reception of a message and the ACK with the RSN included, the receiver should not send messages [22].

It is possible that processes fail while some messages do not yet have their RSNs recorded at the sender. These messages are called partially logged messages. If P2 fails and returns from a checkpoint, it will broadcast requests for its logged messages and the fully logged messages will be replayed in ascending RSN order, starting with the stored RSN+1. Partially logged messages will be sent in any order. As receiver processes cannot send messages while a message is partially logged, no processes other than P2 can be affected by the receipt of a message that is partially logged.

If we consider that P1 fails and retransmits M with the SSN equal to 5, P2 will discard it according to the current SSN value, and if the ACK of this message was not received by P1, P2 will send it.

However, if a process rolls back to a previous state, it will ask all the senders for its logged messages. Thus, the senders would have to stop their executions and look for these messages, unless an FT thread is in charge of managing the message log.

An approximation of the overheads of the protection stage in the a pessimistic SBML approach is represented in Equations 1, 2 and 3, where:

- T_Sender is the time spent by the sender when logging the message.

- $T\_Receiver$ represents the time spent by the receiver when receiving the message.
- $Wait\_ACK$ is the time spent by the process waiting for an ACK.
- I_{Log} and U_{Log} represents the cost of inserting and updating the message log respectively.
- $I_{DataStructs}$ and $U_{DataStructs}$ represent the cost of inserting and updating the data structs in the receiver process. $I_{DataStructs}$ includes the time spent in inserting a new element in the Table of SSN and Table of RSN. $U_{DataStructs}$ represents the time spent in updating the ACK cell in the Table of RSN.

The main objective of these equations is to describe analytically how each message log task may influence the execution time. Equation 1 represents the total possible delay that may be introduced in each message transmission by the protection stage. Considering Equation 2, I_{Log} delays the transmission of each message, $Wait\_ACK$ could be overlapped with computations and U_{Log} penalize the progression of the sender process. However, it is important to note that the times I_{Log} and U_{Log} (Equation 2) spent in the sender (T_Sender) are in its critical path, thus they are forcibly translated into overheads. I_{Log} corresponds to the period of time spent between logical times t_1 and t_2 observed in Figure 2 and U_{Log} corresponds to the period between logical times t_4 and t_5.

Equation 3 represents the time that the receiver is blocked. During this time, the receiver can proceed with its execution but it should not send messages to other processes in order to avoid the creation of orphan processes, because messages are just partially logged.

$$Prot\_SBML = T\_Sender + T\_Receiver \tag{1}$$

$$T\_Sender = I_{Log} + Wait\_ACK + U_{Log} \tag{2}$$

$$T\_Receiver = I_{DataStructs} + Wait\_ACK + U_{DataStructs} \tag{3}$$

It is very difficult to accurately represent the recovery cost of a message logging protocol since it depends on the failure moment, because this affects the re-execution time. In Equation 4 we represent the cost of the recovery stage, taking into account that the receiver has been affected by a failure.

$T\_Restart$ is the time spent in reading the checkpoint from stable storage and restarting the process from it. $T\_Broadcast$ is the time spent in requesting logged messages to all possible senders. N represents the total number of processes, X represents the number of messages that a process has to send to the restarted process (which could be 0 for some processes) and $T\_Message$ is the time spent in sending a message to the restarted process. T_Rex is the time spent in re-executing the process till reaching the pre-fault state.

$$Recovery\_SBML = T\_Restart + T\_Broadcast + \sum_{i=1}^{N} \sum_{j=0}^{X} T\_Message_i + T\_Rex \quad \text{where } i \text{ excludes failed process} \tag{4}$$

It is important to note that the message log is distributed among several nodes, then the complexity when recovering may be higher. The main drawbacks of the SBML are: the complexity in dealing with garbage collection because the message log is distributed among senders; and the disturbances generated to senders when processes are being recovered.

When a process is checkpointed, it should communicate to the senders that messages received before the checkpoint will not be needed anymore. This could be a costly operation if there are many senders and a broadcast operation is used. However, some works like [13] propose to piggyback this information when sending an ACK to a sender after a checkpoint.

In faulty scenarios, the failed processes should ask for their logged messages to all possible senders. The senders should stop their execution, or use a separated thread, to serve the logged messages to the failed processes, thus the recovery of a process affects the execution time of non-failed processes.

3.4. Pessimistic Receiver-Based Message Logging

In RMBL protocols, the receiver is in charge of logging each received message into a stable storage. Thus, in the event of failure, processes are able to reach the same before fault state by reproducing in order the non-deterministic events logged.

The pessimistic RBML [4] is a solution that introduces more overhead during failure-free executions because each received message should be retransmitted to a stable storage (eg. memory buffer in another node.). This

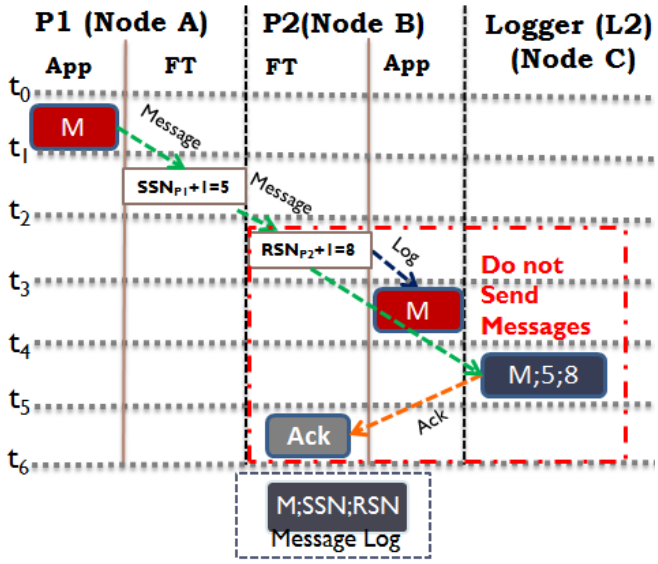


Figure 3: Receiver-Based Message Logging during failure-free execution.

solution may introduce overheads in communications and also in computations if there are not dedicated resources to deal with message logging.

The main idea behind RBML is to allow failed processes to recover faster, by avoiding message requests to non-failed processes, and also simplify the garbage collection, since after a process checkpoint all its received messages can be erased from the log which is saved only in one location. Moreover, the recovery is a local task, then a process do not depend on several processes to reach the before-failure state. In the pessimistic version of this logging protocol, the receiver processes should not send any messages to other processes while all the previous received messages are not properly saved into stable storage. This is done like this in order to avoid the creation of orphan processes in case of failure.

In Figure 3 we illustrate the RBML operation in its pessimistic version. Here we are considering that there are three processes involved, the sender P1, the receiver P2 and the logger of P2 which is L2. In a system where there are not dedicate resources, application processes will compete for resources with the fault tolerance processes and the logger processes. In Figure 3 we are showing in each process and node only the parts involved. We are also assuming that the message M contains all the header information about destination and source.

As can be observed in Figure 3, P1 sends a message M with its current

SSN, P2 receives M, appends its RSN to it and sends it to a stable storage (L2) (SSN and RSN tables are saved together with each checkpoint). Message M is also delivered to the application, it is important to highlight that operations between instants t_3 and t_5 can be overlapped. Then, P2 waits for the confirmation of L2, communicating that message M is properly saved in order to allow the application to send new messages. This avoids the impact of partially logged messages in other processes besides the receiver, because if P2 sends and confirms the logging of a message M1 to another process P3 but fails without logging M, then P3 will be an orphan process.

Let us suppose that P2 fails and restarts from a previous state. After restarting, P2 will transfer the message log from L2 to its new local node and consume it in order to reach the pre-fault state. As the non-failed processes that have received a message from P2 have the SSN value of P2, they can easily discard messages that P2 sends during recovery.

The cost of the protection stage in the pessimistic RBML approach is represented in Equation 5, where $T\_Forward\_M$ is the time spent in forwarding the received message to a stable storage, such as memory of another node. Then the receiver should wait for the insertion of the message into the message log (I_{Log}) and finally wait for the ACK ($Wait\_ACK$) that indicates that the message is properly saved. During these periods, the receiver should not send messages to other processes in order to avoid the creation of orphan processes. We consider that times spent in increasing values of SSN and RSN are negligible, since there is no need to insert values in special data structures.

$$Prot\_RBML = T\_Forward\_M + I_{Log} + Wait\_ACK \tag{5}$$

In Equation 6 we represent the cost of the recovery stage, taking into account that the receiver process has been affected by a failure. $T\_Restart$ is the time spent in copying the checkpoint from stable storage and restarting the process from it. T_Log represents the time spent in transferring the message log saved in stable storage (memory of other node, hard disk, etc.), and T_Rex represents the time spent in re-executing the process till reaching the pre-fault state, the re-execution is usually faster than normal execution since the restarted process has received all messages locally.

$$Recovery\_RBML = T\_Restart + T\_Log + T\_Rex \tag{6}$$

In order to erase old messages, after a process finish its recovery could make a checkpoint and delete all logged messages. This time could also be used to calculate the recovery time.

If we compare Equation 1 with Equation 5, we can observe that main difference resides in the time that the RBML approach should spend in forwarding each received message. In SBML, if the receiver can proceed with computation while the message logging phase is taking place, it would not perceive a high delay.

If we compare Equation 4 with Equation 6, we can observe that when using RBML, the restarted process will be able to reply its messages independently, relaying only on the messages saved in the log. On the other hand, when using SBML, the restarted process will need to consume a message log which is distributed among several senders, and this may lead to higher complexity of the recovery procedure. When collective operations are used or non-failed processes are disturbed during their normal execution, these procedures may lead to higher recovery times. It is also important to highlight that in RBML, the garbage collection is less complex than in SBML. In RBML, when a process is checkpointed it can send a single command to delete all its saved log. On the other hand, when using SBML and a process is checkpointed, all senders should be notified and messages sent to the process before the checkpoint operation should be deleted.

4. Hybrid Message Pessimistic Logging

The Hybrid Message Pessimistic Logging HM_{PL} aims to reduce the overhead introduced by pessimistic receiver-based approaches by allowing applications to continue with their normal execution while messages are being fully logged. In order to guarantee that no message is lost, a pessimistic sender-based logging is used to temporarily save messages while the receiver fully saves its received messages. The HM_{PL} manages messages using a distributed controller and storage to avoid harming system's scalability.

4.1. Key Concepts

The HM_{PL} could be presented as a combination of a pessimistic SBML and an optimistic RBML. We have designed the HM_{PL} to guarantee that no message is lost in presence of failures in order to allow failed processes to reach the same before-failure state. In order to design and develop the HM_{PL} we have set these main objectives:

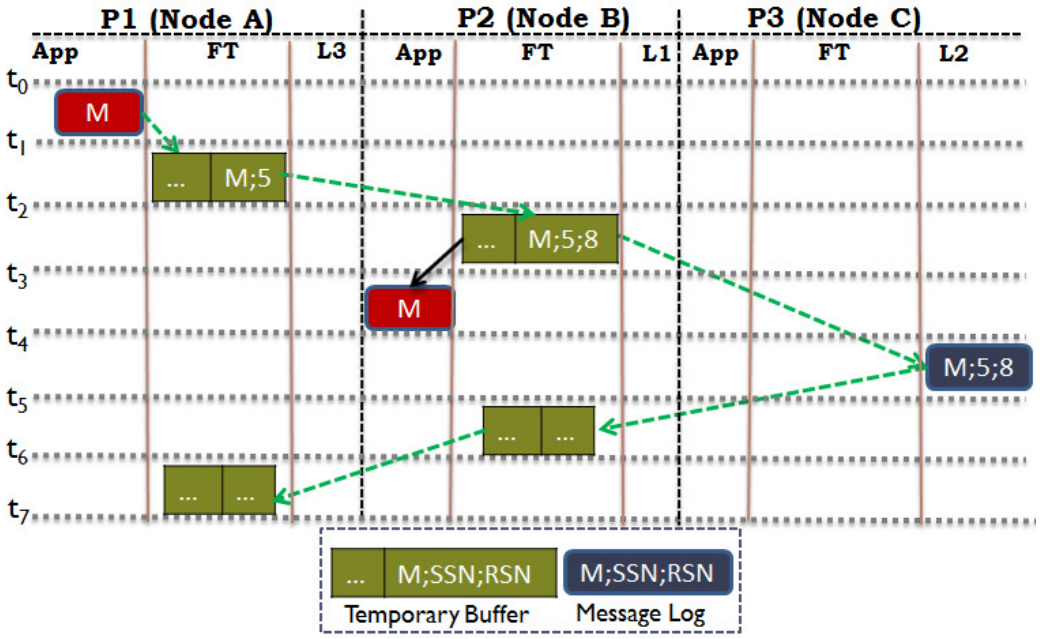


Figure 4: Hybrid Message Pessimistic Logging. Event ordering and redundant information during failure-free executions. Green dotted-lined arrows represent logging operations and black arrows represent application messages.

1. Availability: we focus on providing a strategy that could achieve a MTTR of processes similar to those obtained when using a RBML approach. In order to achieve this, we focus on maintaining the fast recovery feature of the RBML by allowing a process to restart and continue with its execution without disturbing non-failed processes.

2. Overhead Reduction: the overheads in communication times during failure-free executions introduced by a RBML technique could be very high [4]. Another source of overhead comes from the blocking behavior of the pessimistic version of RBML. We have focused on removing these blocking phases from the critical path of a parallel application so we can reduce the overhead introduced.

4.2. Design

As mentioned before, RBML approaches allow fast recovery of failed processes (low MTTR) and SBML approaches introduce low overhead during failure-free executions. Taking into this account, we designed the HM_{PL} by

combining both strategies. The best way to maintain a low MTTR when using a message log is to save messages when receiving them, so the message log is not distributed among several processes. By saving received messages, failed processes may restart from a previous state and then consume this message log without broadcasting requests for past messages to non-failed processes. The problem with pessimistic $RBML$ is that each received message should be forwarded to stable storage (eg. memory of other node) and this increases the message transmission and management times.

On the other hand, SBML approaches introduce less overhead during failure free executions since messages may be saved in a buffer of the sender. If a process fails and restarts, it has to ask for all necessary messages to the senders in order to reach the pre-fault state. As we have seen, garbage collection is more complex in SBML since a checkpointed process should notify the senders so they can erase old messages that belong to the checkpointed process. It is also important to highlight that the execution of logging tasks consumes CPU cycles, and this may impact on the execution time if there are not dedicated resources for the FT tasks.

Figure 4 shows the basic operation of the HM_{PL} when messages are sent from one process to another (P1 to P2). We have introduced a new data structure which is a temporary buffer. This buffer is used in the sender and also in the receiver to temporally save messages in order to allow the receiver to communicate without waiting for the message log protocol to finish the full cycle. Thus, the HM_{PL} removes the blocking behavior of SBML and RBML while the logging is taking place, which means that receiver processes can communicate with others while messages are just partially logged. The logger L2 stores the messages in an array in memory which is flushed to disk asynchronously when a memory limit is exceeded or opportunistically. As we have seen, message M has all the information about sender and receiver.

Figure 5 shows the flow diagram of the HM_{PL} during the protection stage. It is important to highlight that we assume the utilization of a transparent fault tolerant middleware that intercepts and manages messages (FT column in the figure). Before sending a message, FT of P1 inserts the message M with its SSN in its temporary buffer (TB). When FT of P2 receives M (checking the SSN to discard already received messages), it inserts M, the SSN and RSN in the TB and proceeds with the normal execution. In the meantime, the FT process of P2 will transmit the message and all the extra data to L2, which is located in another node, and once FT of P2 receives the confirmation that M has been correctly saved, it will erase M from its TB (t6 in Figure 5)

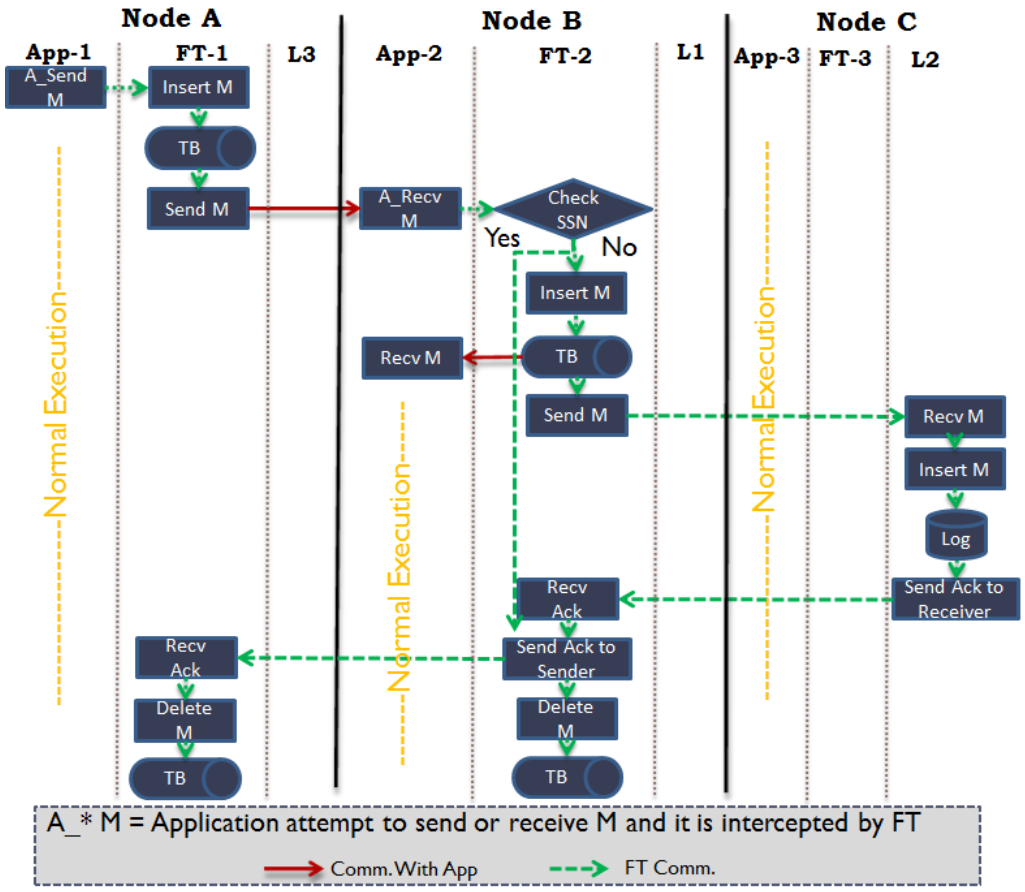


Figure 5: Hybrid Message Pessimistic Logging. Protection mechanism and main tasks considering three nodes.

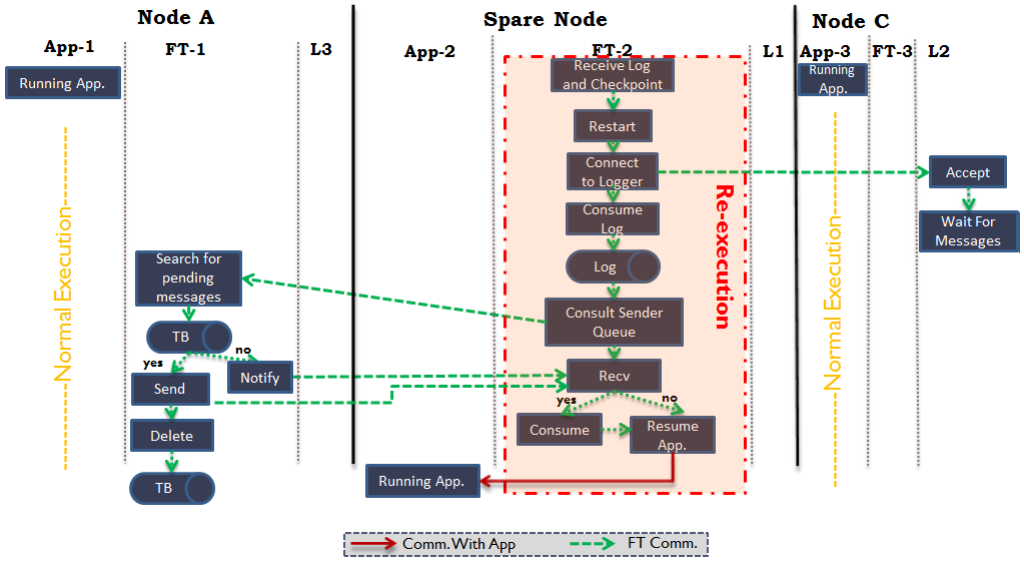


Figure 6: Hybrid Message Pessimistic Logging. Recovery of the receiver.

and inform FT of P1 so it can erase M also from its TB (t7).

Figure 6 illustrates the flow diagram of the recovery procedure of the receiver. FT of P2 will receive the message log and checkpoint and will restart the application, connect to its new logger and consume the message log saved by the Logger L2. After finishing this, FT of P2 will ask P1's FT if it has a message in its TB and consume it. Then, the normal execution will continue, but when P2 has to receive the first message after recovery from a sender, P2's FT will ask if there are not messages in the sender's TB.

It is important to highlight that in most cases the receiver will be able to fully recover using its message log. However, when there are partially logged messages, it will need to ask for messages in the TBs of senders. The senders will hardly be affected by requests since the receiver may directly access the sender's TB through the fault tolerance middleware and copy messages.

Taking into account the sender side of each process, when a process is restarted from a previous checkpoint it may re-send some messages that the FT process will transparently discard according to the current SSN that the receiver has for that sender. Furthermore, the receiver can inform the sender of the current value of the SSN, so the sender will avoid resending unnecessary messages.

Let us analyze the failure moments and what the impact is on each situation (we focus on the failure of the receiver). In order to explain the failure

moments, we will use the logical times (t_x) of Figure 4:

- *t2-t4*: P1's FT has the message in its TB, so once P2 starts re-executing it will consume all saved log in L2, and then ask P1 for message M.
- *t4-t6*: P1's FT has the message in its TB. L2 has also saved M but it does do not send the ACK to P2, so L2 will erase this message and P1 will re-send M to P2 during re-execution.
- *From t6*: M has been saved in L2 and confirmed to P2's FT. P2 will consume this message from the message log. In situations where M has not been erased from P1's queue, after restarting P2 will continue with the logging procedure where it stopped and will send the ACK to P1 so it will remove M from its TB.

The overhead of the protection stage in the HM_{PL} is modeled in Equation 7, where I_{TB} is the cost of inserting each message in the TB of the sender or the receiver. Equation 8 shows the times that are not in the critical path of each message transmission (unlike the RBML) when using HM_{PL}, where $T\_Forward\_M$ is the time spent when retransmitting the message M to the Logger in another node, I_{Log} is the time spent in introducing the message in Log and $Wait\_ACK$ is the time spent in waiting for ACKs from other nodes. We are not taking into account the times spent on increasing SSN and RSN values or checking them, since these times could be negligible.

$$Prot\_HM_L = 2 * I_{TB} + T\_Overlapped \tag{7}$$

$$T_{Overlapped} = T\_Message + I_{Log} + 2 * Wait\_ACK \tag{8}$$

In Equation 9 we represent the cost of the recovery stage, where $T\_Restart$ is the time spent copying the checkpoint from another node and restarting the process from it, T_Log represents the time spent copying the message log from stable storage (e.g. memory another node). M represents the number of neighbors (senders) that have a logged message for the restarted process in their TB. $T\_Message$ is the time spent copying messages from senders. T_Rex represents the re-execution time and $T\_ConnectLogger$ is the handshake time between the restarted process and the Logger.

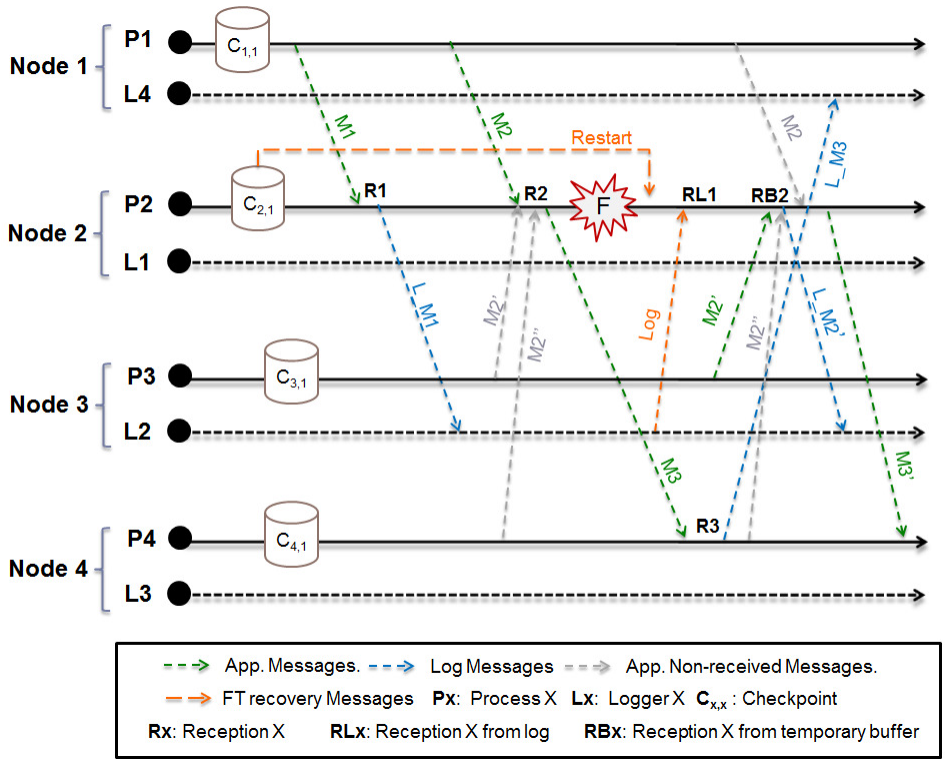


Figure 7: Orphan Processes in the Hybrid Message Pessimistic Logging. P2 is restarted in a spare node after the failure.

$$Recovery\_HM_L = T\_Restart + T_{Log} + \sum_{i=1}^{M} T\_Message_i + T\_Rex + T\_ConnectLogger \quad (9)$$

4.3. Orphan Processes

As the HM_{PL} combines a pessimistic SBML with an optimistic RBML, there may be a possibility of creating orphan processes. Here we explain how we avoid the creation of orphan processes. In message logging, the order of reception is considered the unique source of non-determinism. When using MPI to write parallel applications, the main source of non-determinism is the usage of the wild card MPI_ANY_SOURCE to receive a message. This tag allows the reception of messages from any possible sender, without specifying a particular one. In the next paragraphs when we mention Mx' or Mx" we

are talking about a message Mx that is coming from another destination.

Let us consider the situation in Figure 7, where message M1 is sent from P1, received in P2 (R1) and logged in L2 (L_M1). Assuming that reception R2 in P2 is done with a wild card, then P2 receives M2 from P1. Suppose that having received the message M2, P2 sends M3 to P4 (R3) and then fails without logging M2 in its logger.

P2 restarts from its checkpoint ($C_{2,1}$) and reads M1 from its log (RL1) and, instead of consuming M2 from P1's temporary buffer, it consumes M2' from the temporary buffer of P3. Then, P2 saves M2' in its logger (operation L_M2') and produces M3' instead of M3 making P4 an orphan process.

In order to avoid the situation explained above, when we detect that a wild card reception is being used, we do not allow the receiver to continue till the message is fully logged. This also allows us to avoid the generation of requests to all processes' temporary buffers when restarting.

As different senders have probably initiated the communication to a wild card reception (M2, M2' and M2"), non-confirmed messages by the receiver will be erased from the TB of the senders after the receiver process confirms the reception of a subsequent message.

4.4. Implementation Details

The proposals presented in this work has been included inside the RADIC architecture. RADIC is fault tolerant architecture based on a rollback-recovery protocol which uses a pessimistic receiver-based message logging associated with uncoordinated checkpoints. RADIC does not need any coordinated or centralized action or element to carry out their fault tolerance tasks and mechanisms, so application scalability depends on itself. RADIC acts as a transparent fault tolerant layer between the MPI standard and the parallel machine, providing a fault resilient environment. The scalability of RADIC architecture has been addressed in [9] [16]

RADIC operations can be observed in Figure 8. All message exchanges are made through components called Observers. After receiving a message, an Observer forwards this received message to a component residing in another node called Protector (Figure 8a). Protectors store checkpoints of the processes that they protect and they have logger threads that are in charge of storing received messages. A node failure can be detected through a Heartbeat/Watchdog mechanism or by Observers that try to communicate with a failed process (Figure 8b). Failures are masked in order to avoid a fail-stop behavior.

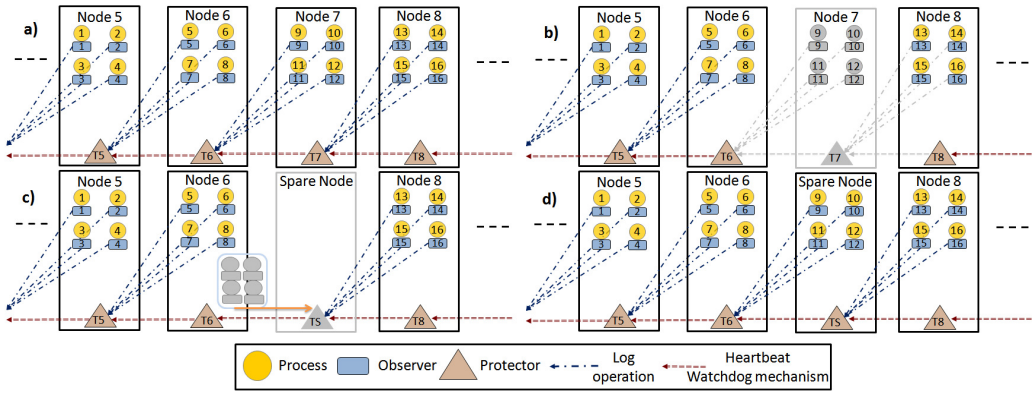


Figure 8: RADIC Scenarios: a) Fault free execution. b) Failure in Node 7. c) Inclusion of Spare Node , transference of checkpoints, Heartbeat/watchdog restoration and assignation of a new protector to processes of Node 8. d) Restart of faulty processes in Spare Node.

In order to maintain the computational capacity and the initial tuning in presence of node failures, failed processes may be automatically restarted in Spare Nodes [23] (Figure 8c). As can be observed in Figure 8d, after the recovery phase takes place, the initial configuration is maintained.

We have included the HM_{PL} inside the RADIC-OMPI implementation [23]. The HM_{PL} has been included in the *Vprotocol* Framework of Open MPI, since this framework enables the implementation of new message logging protocols in the Open MPI library [14]. The main components of our message logging implementation are described below. As the HM_{PL} is a combination of sender and receiver approaches, we split the functionality to explain it:

1. *Sender Message Logging*: Before sending a message, the payload of the message is saved in a circular queue (temporary buffer) in memory. As this circular queue will be continuously modified, there is no need to flush it to disk. Before sending another message to the same receiver, the non-finished logging transmissions will be checked in order to erase messages from the circular queue. Normally, the circular queue will contain at most one message per receiver. However, in order to guarantee that the size does not grow uncontrollably, the circular queue size is limited according to a percentage of the total memory available in the node.

2. *Receiver Message Logging*: When a message is received, it is introduced

inside a circular queue and then the message is sent to the logger residing in another node by using a non-blocking communication. When the logger informs that this message has been saved it will remove it from its circular queue. Before receiving a message the SSN is always checked in order to discard already received messages.

3. *Logger*: We have added special threads (one per application process) that are executed outside the communicator of the parallel application. Each logger publishes its name, so an application process can get connected to a logger residing in a different node when finishing the MPI_Init command . When a message is received from a connected process, the logger will save this message in its volatile memory until a defined level of memory is consumed, then it will asynchronously start to flush data to disk or it can also command a checkpoint because a memory limit was exceeded.

5. Experimental Validation

The experiments to test our method have been carried out using a Dell PowerEdge M600 with 8 nodes, each node with 2 quad-core Intel® Xeon® E5430 running at 2.66 GHz. Each node has 16 GB of main memory and a dual embedded Broadcom® NetXtreme IITM 5708 Gigabit Ethernet. RADIC features and the message logging techniques have been integrated into Open MPI 1.7. The RADIC middleware is the one in charge of all message transactions that are made through the MPI library. Then, all messages are intercepted and treated according to the specification of each described message logging technique.

In order to evaluate the HM_{PL}, we have compared it with a pessimistic RBML since the main objective of the proposed HM_{PL} is to reduce the overhead of classic RBML avoiding a high penalization in case of failure.

5.1. Initial Validation

In order to make the first experimental validation of our HM_{PL}, we have used the NetPipe tool [24]. We compare the HM_{PL} with a classic pessimistic receiver-based message logging technique. We have executed the Netpipe tool using two processes in two different machines, where one of the process acts as a sender and the other as a receiver. When using a message logging approach, messages are retransmitted to a third process running in a third machine.

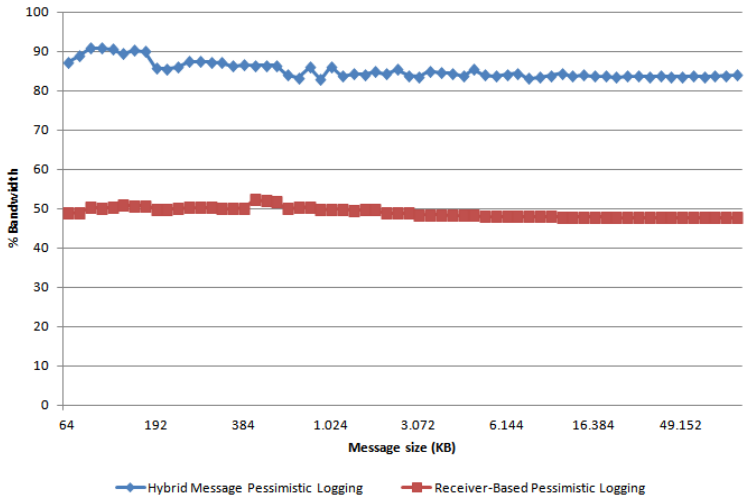


Figure 9: Bandwidth utilization obtained with NetPipe Tool. 100 % of bandwidth is achieved by not using any log mechanism.

When using the receiver-based logging, every time a message is received, it is first forwarded to a logger thread residing in another node, and only after the message is fully saved does the MPI receive call finalize. On the other hand, as the HM_{PL} proposes the utilization of temporary buffers, there is no need to wait for messages to be saved while they are being transmitted to logger threads. Thus after receiving each message, it is copied to a temporary buffer and then the MPI receive call can finalize. Ideally, the temporary buffers would contain at most one message per each peer process. However, when messages are produced faster than they are consumed, the temporary buffers may increase in size. In order to avoid uncontrolled grows in size, we set a memory limit for each temporary buffer according to systems' memory and once this threshold is surpassed, the HM_{PL} would act as a pessimistic RBML waiting for confirmation of received messages (Figure 3).

Figure 9 and Figure 10 show the bandwidth utilization and the overheads respectively (measured by the NetPipe tool), when using each of the message logging techniques described in failure-free executions. We should notice here that the benchmark executed without message logging represents 100% of bandwidth utilization and overhead of 0 % (yaxis). Moreover, a bandwidth value of 100 % is achieved when no message logging techniques are being used. Lower bandwidth values indicate a penalization due to re-transmission of packets (message logging) .The overhead is calculated by comparing the transmission time of HM_{PL} and receiver-based logging with the transmission

time of packets when not applying any log mechanism. We are showing results between 64 KB and 64 MB message sizes.

As the default eager limit of the MPI library is set to 64 KB, for messages smaller that this size there is no difference between both message logging techniques. The eager limit is the max payload size that is sent with MPI match information to the receiver as the first part of the transfer. If the entire message fits in the eager limit, no further transfers / no Clear-to-Send (CTS) are needed. If the message is longer than the eager limit, the receiver will eventually send back a CTS and the sender will proceed to transfer the rest of the message.

In Figure 9 can be observed that the bandwidth utilization with the HM_{PL} remains close to 85% while with the receiver-based logging only reaches 50%. Figure 10 shows that the overheads with the HM_{PL} are near to 20% for each message transmission and with the receiver-based logging it is near to 100%, since for each sent message the application should wait double the time for an answer.

It is important to highlight that these results are just taking into account two independent processes running in two different nodes, and one logger thread in other node. Moreover, the system is not under full utilization and the network cards are not overloaded. In addition, the NetPipe tool just gives us an approximation of the logging overheads taking into account the communication times but not the computation cost of logger threads. Thus, the added overhead will also depend on applications behavior and usage level of the system.

5.2. Comparison of Logging Techniques in Failure-free Executions

In these experiments we measure the impact of each message logging technique without taking into account the impact of checkpoints. The main objective of these experiments is to analyze the impact of two message logging techniques (HM_{PL} and Pessimistic RBML) assuming the usage of two different checkpoint strategies: Uncoordinated Checkpoint and Semi-Coordinated Checkpoint [20]. When the strategy used is uncoordinated checkpoint, all message transmissions are logged in a different node. When using a semi-coordinated strategy, only message transmissions that go from one node to another are saved in a logger residing in another node. The time required to take semi-coordinated checkpoints may be longer than uncoordinated checkpoints since coordination between processes in the same node is needed, but this impact is not analyzed here.

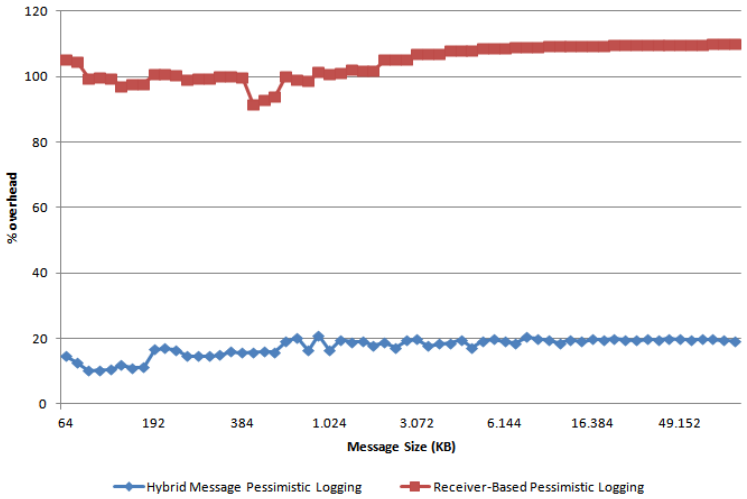


Figure 10: Overheads in message transmissions obtained with NetPipe Tool. 0 % of overhead is achieved by not using any log mechanism.

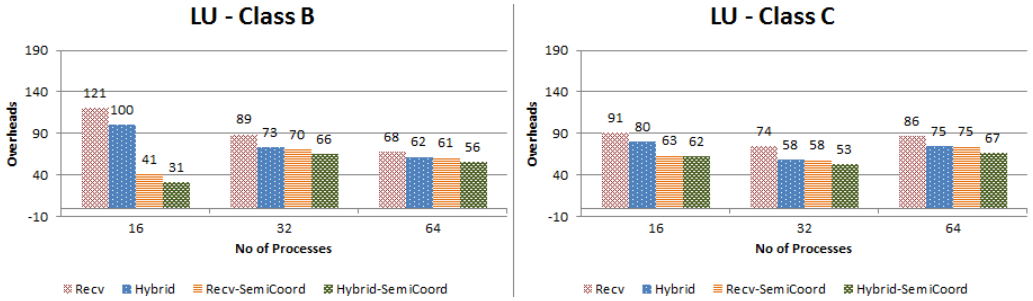
In these experiments, the logger threads share cores with application processes, so there is also an impact in computations but that is homogeneously distributed among processes by using CPU affinity to attach each logger to a core. When there are free cores in a node, these cores are not used by application processes neither by fault tolerance processes. So, it is ensured that there are not dedicated resources to fault tolerance threads.

As testbeds we have used the LU, CG and BT benchmarks from the Nas Parallell Benchmarks (NPB) suite with classes B and C [25]. Each result presented represents a mean of 5 different and independent executions with a variance between 1 and 5%.

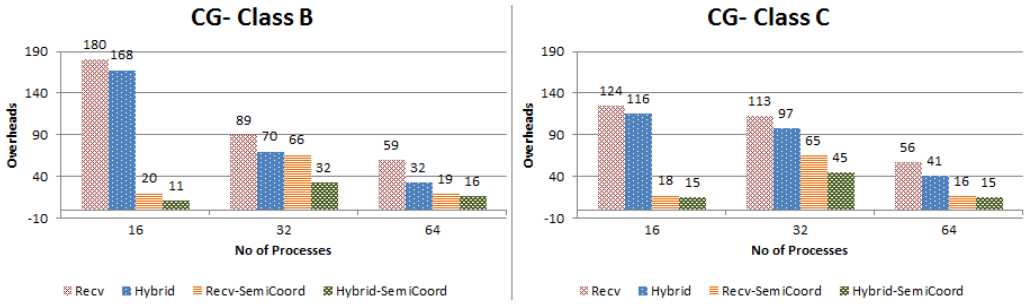
In Figure 11, we summarize all results obtained by comparing the Hybrid Message Pessimistic Logging (HM_{PL}) approach with a Pessimistic Receiver-based Message Logging (RBML) approach. Below we explain each type of execution made:

- *Recv:* Executions made using the default pessimistic RBML mechanism of RADIC. All messages are saved in a logger residing in another node, even messages between processes residing in the same node. We are assuming the usage of a fully uncoordinated checkpoint approach.

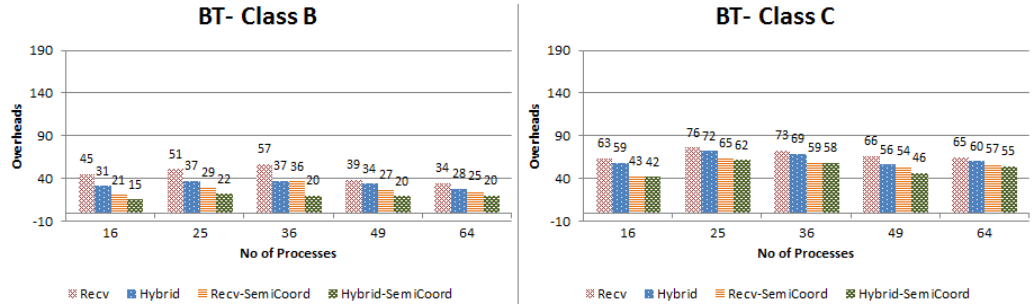
- *Hybrid:* In these executions we are using the Hybrid Message Pessimistic Logging (HM_{PL}) proposed in this paper. We are assuming the usage of a fully uncoordinated checkpoint approach.



(a) Overhead comparison using LU benchmark with classes B and C.



(b) Overhead comparison using CG benchmark with classes B and C.



(c) Overhead comparison using BT benchmark with classes B and C.

Figure 11: Comparison of overheads using the Hybrid Message Pessimistic Logging and the Pessimistic RBML considering the NAS Parallel Benchmarks during failure-free executions. Two checkpoint protocols are used: uncoordinated and semi-coordinated at node level.

- *Recv-SemiCoord:* Executions made using the default pessimistic RBML mechanism of RADIC. In this case we are assuming the usage of the Semi-coordinated Checkpoint of RADIC Architecture [20].

- *Hybrid-SemiCoord:* Executions made using the HM_{PL} using the Semi-coordinated checkpoint approach of RADIC.

According to the results shown in Figure 11, we can observe that the HM_{PL} reduces overheads in all cases tested. In each column we can observe the overhead introduced by each message logging technique used in comparison to the execution without using message logging. For this set of experiments, we have used a *fill-up* strategy to map processes in each node. Nevertheless, as RADIC requires at least 3 nodes to work properly, experiments with 16 processes have been made with 3 nodes with 7 processes in the first and second node and 2 processes in the third. It is important to mention that when there are free-cores in one machine, they are not used by application processes neither by fault-tolerant threads. This allows us to take into account the overhead in computation as well as the overhead in communication of each approach.

Figure 11a shows the overheads obtained when executing the LU benchmark. When using the uncoordinated approach of RADIC, we can observe an overhead reduction between 6% (Class B, 64 processes) and 21% (Class B, 16 processes). If we use the semi-coordinated checkpoint option of RADIC, we can observe an overhead reduction between 1% (Class C, 16 processes) and 10% (Class B, 64 processes).

Figure 11b shows the overheads obtained when executing the CG benchmark. When using the uncoordinated approach of RADIC we can observe an overhead reduction of 8% in the worst case scenario (Class C, 16 processes) and 27% (Class B, 64 processes) in the best case. If we use the semi-coordinated checkpoint option of RADIC we can observe an overhead reduction between 1% (Class C, 64 processes) and 34% (Class B, 32 processes).

Results obtained with the BT benchmark are presented in Figure 11c. With the HM_{PL} and the uncoordinated checkpoint approach of RADIC, we can observe an overhead reduction of 4% in the worst case scenario (Class C, 25 processes) and 20% (Class B, 36 processes) in the best case. If we use the semi-coordinated checkpoint option of RADIC, we can observe an overhead reduction between 1% (Class C, 36 processes) and 16% (Class B, 36 processes).

The obtained results show that the HM_{PL} is able to reduce overheads in parallel applications during failure-free executions. The experiments presented in this section take into account impact in communications as well as in computations. In order to take into account the impact in communications we have forced that application processes and fault tolerance threads compete for resources in each core. In the next subsection we will evaluate the

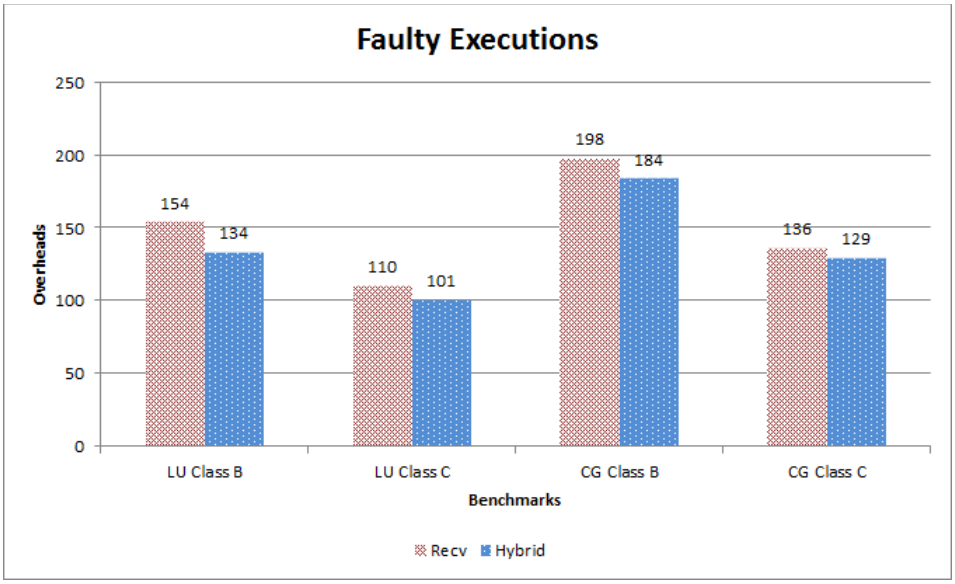


Figure 12: Total execution time overhead comparison between the Pessimistic Receiver-based Message Logging and the Hybrid Message Pessimistic Logging in executions affected by 1 failure.

recovery times of the HM_{PL} and compare them with a pessimistic RBML.

5.3. Experimental Results in Faulty Executions

Here we analyze and compare the impact of the the Hybrid Message Pessimistic Logging (HM_{PL}) approach with a Pessimistic Receiver-based Message Logging (RBML). The main objective of these experiments is to show that the HM_{PL} is able to reduce overheads in the total execution time in applications affected by failures. The experiments presented here are executed using 3 nodes and 16 processes, the first two nodes with 7 processes and the third one with 2 processes. When there are free cores in a node, these cores are not used by application processes neither by fault tolerance processes (CPU affinity is used to this purpose). So, it is ensured that there are not dedicated resources to fault tolerance threads. The uncoordinated checkpoint approach of RADIC is used. Restart and recovery are made using a spare node.

In these experiments we consider that the applications are divided in events, where an event in this case represents the reception of one message in one process. Checkpoints are taken in the same event in each pair of executions. Failures are also injected in the same event, in order to properly

	Benchmark							
	LU Class B		LU Class C		CG Class B		CG Class C	
FT Technique	**RBML**	**HM_{PL}**	**RBML**	**HM_{PL}**	**RBML**	**HM_{PL}**	**RBML**	**HM_{PL}**
Property								
Processes	16	16	16	16	16	16	16	16
Checkpoint Event	32148	32148	60405	60405	5340	5340	6993	6993
Failure Event	32447	32447	60884	60884	5522	5522	7176	7176
Message Log Size (KB)	348	348	836	836	12288	12288	23552	23552
Checkpoint Size (MB)	101	101	153	153	93	93	133	133
Pre-checkpoint Time (Sec.)	52,05	46,89	165,29	156,05	171,81	166,00	324,60	316,54
Checkpoint Time (Sec.)	9,48	9,50	17,63	17,64	10,40	10,43	14,68	14,67
Checkpoint Transference Time (Sec.)	2,66	2,60	3,60	3,60	2,53	2,64	3,23	3,35
Time to failure after checkpoint (Sec.)	0,27	0,23	0,93	0,88	1,64	1,34	3,70	3,27
Checkpoint and Message Log to spare Time (Sec.)	3,54	3,49	4,48	4,50	3,62	3,71	4,52	4,64
Checkpoint Restart Time (Sec.)	1,28	1,14	3,12	3,83	1,12	1,20	1,87	2,10
Log Comsumption Time (Sec.)	0,20	1,08	0,78	1,69	0,42	0,44	1,11	1,19
Time from restart to end (Sec.)	71,77	65,01	165,09	156,82	278,54	262,23	327,33	315,33
Total Execution Time (Sec.)	141,26	129,94	360,92	345,01	470,09	447,99	681,05	661,08
Execution Time Without FT (Sec.)	55,55	55,55	171,50	171,50	157,59	157,59	288,10	288,10
Total Overhead (%)	**154,3**	**133,9**	**110,4**	**101,2**	**198,3**	**184,27**	**136,4**	**129,5**

Figure 13: Breakdown of Recovery Times in executions affected by 1 failure. Failure affects node 3 which has 2 processes running in it.

compare each pair of executions. Failures are injected in the third node where two processes are residing. One spare node is used to restart failed processes.

In the executions made, the overheads in computations are homogeneously distributed among processes by using CPU affinity to attach each logger thread to a core.

In Figure 12 we summarize the results obtained by comparing the total execution time with the HM_{PL} approach and with a Pessimistic RBML approach considering a single failure. Figure 13 presents a breakdown of times and details about failure injection. Figure 12 summarizes the overheads in faulty executions, using the LU and the CG benchmarks with classes B and C. With the class B of the LU benchmark, the HM_{PL} introduces 20% less overhead than the RBML and 9% less overhead with the class C. Considering the class B of the CG benchmark, the HM_{PL} introduces 14% less overhead than the RBML, and 7% less considering class C.

In order to extract the measures shown in Figure 13, we have inserted timers inside the RADIC architecture that allow us to determine time consumption of each step of the executions. The message log sizes and checkpoint sizes that are shown have been calculated considering one process when these

data is transferred to the spare nodes. Below we explain some of the rows of Figure 13 that are not self-explanatory:

- *Checkpoint Event*: message number where the checkpoint is triggered.
- *Checkpoint Size*: the checkpoint size includes the overhead in size caused by temporary buffers. As it can be observed this overhead does not affect the checkpoint size (taking into account memory size in order of MB).
- *Failure Event*: message number where the failure is injected.
- *Pre-checkpoint Time*: represents the time spent from the beginning of the application till the checkpoint takes place.
- *Time to failure after checkpoint*: represents the time elapsed between the checkpoint and the failure injection.
- *Checkpoint and Message Log to spare Time*: is the time spent in transferring data to the spare node.
- *Checkpoint Restart Time*: is the time spent in restarting the process from checkpoint.
- *Log Consumption Time*: represents the time spent in reading messages from the message log. When using the HM_{PL}, it also includes the time spent in copying messages from the temporary buffer of senders.
- *Time from restart to end*: is the time elapsed between restart finalization and application ending.

The main difference between the pessimistic RBML and the HM_{PL} can be observed in the *Log Consumption Time* row, since when using the HM_{PL}, failed processes should ask some messages to their senders and this causes an almost negligible overhead.

Recovery times of the BT benchmarks are not presented here because the current RADIC implementation is not able to re-execute properly failed processes that belong to communicators created with the MPI_Comm_split command. The inclusion of restarted processes inside a created communicator will be addressed in the future.

In faulty executions the HM_{PL} maintains almost the same complexity of the RBML (garbage collection is simple), slightly increasing the time to consume the message log. However, this has an almost negligible impact on the parallel execution, thus the time spent in recovery of the RBML and the HM_{PL} are almost the same. Therefore, the HM_{PL} seems to be a suitable replacement to pessimistic RBML, since it reduces the overhead in failure-free executions with a negligible impact in recovery times.

5.4. Limitations and Overhead Analysis

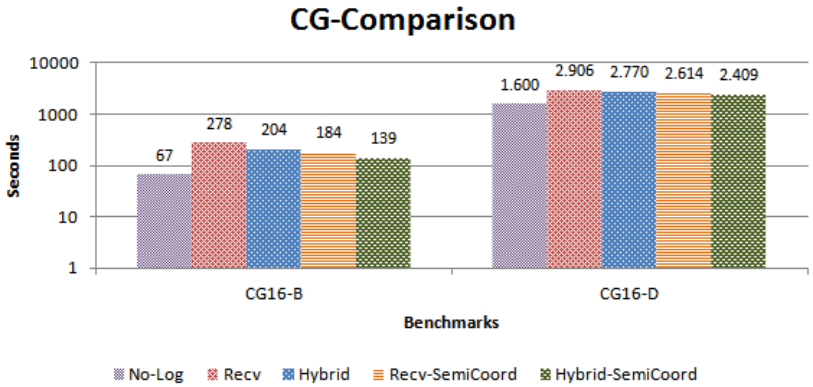
According to what has been demonstrated in the previous experiments, the HM_{PL} introduces less overhead than the pessimistic RBML. However, the overheads in some situations and for some specific configurations could be high and may discourage the usage of message logging.

Here we are going to discuss the limitations of the HM_{PL} and analyze possible causes of overheads. Specifically, we will consider the CG benchmark (Figure 11b), which presents high overheads in some scenarios. The aim of this discussion is to discover the bottlenecks that avoid the HM_{PL} to perform better and analyze the best case scenarios for this message logging technique.

Figure 14 illustrates the execution of the CG benchmark class B and D. In these executions we have put 2 processes per-node, using a total of 8 nodes in order to avoid system overload, so the applications could be analyzed easily. As can be observed, overheads in class D are lower than in class B. The principal reason for this is that the execution of class B in our execution environment (cluster) is communication-bound.

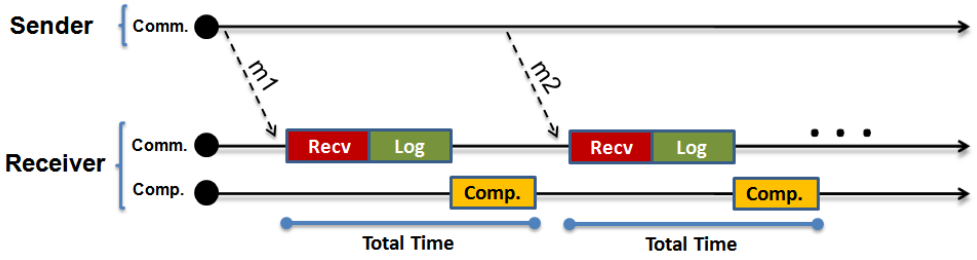
We have analyzed the execution of each of these benchmarks with the PAS2P tool [26]. We find that the communications represent 82% of the execution time of the CG class B with 16 processes. Then, any disturbance introduced in the communications will considerably affect the total execution time. Considering the CG class D with 16 processes, the communication time represents 36% of the application, therefore this is a computation bound scenario.

The HM_{PL} focuses on removing blocks from the critical path of the application. However, extra internode messages will be created, and if these transmissions could not be overlapped with computations, there is no way to hide overheads. It is important to note that in communication-bound applications, the overheads will be considerable since the HM_{PL} will not be able to overlap the logging step with computation.

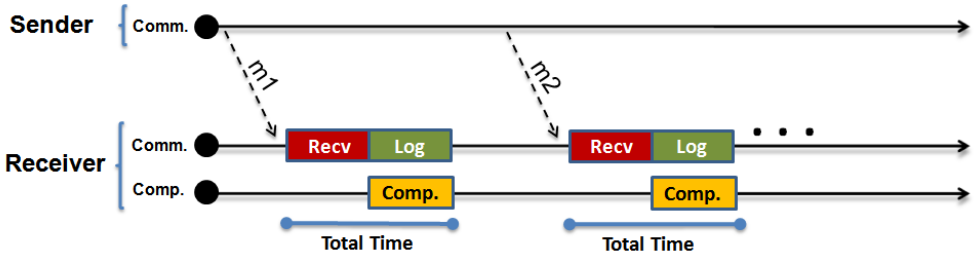


Overheads (%)		
Logging Types	CG16-B	CG16-D
Recv	316,89	81,64
Hybrid	206,38	73,11
Recv-SemiCoord	175,97	63,40
Hybrid-SemiCoord	108,73	50,55

Figure 14: Overhead Analysis of the CG benchmark.



(a) Operation with the Receiver-Based Message Logging.



(b) Operation with the Hybrid Message Pessimistic Logging.

Figure 15: Synthetic Application Operation.

In computation bound scenarios, message logging approaches will be able to hide a percentage of the overheads in communications with the computa-

tions. Taking into account that the HM_{PL} focuses on removing blocks from the critical path of applications, this provides a better chance to overlap the logging operation with computations.

The HM_{PL} is able to considerably reduce the overheads when comparing it with the RBML, as have been seen in subsection 5.1. However, if we consider the computation bound scenario of the CG class D in Figure 14 as an example, the HM_{PL} is able to reduce around 13% of overhead in comparison with the RBML. Therefore, it is important to analyze the possible causes that prevent better results.

We have first analyzed the best scenarios for the HM_{PL}, in order to find out what the maximum improvement that could be achieved is. A synthetic application has been developed in order to determine the maximum potential of the HM_{PL}. In this synthetic application we have a process that sends messages of different sizes to a receiver, and after the reception of each message the receiver does some computation. Figure 15a illustrates the behavior of the synthetic application when using a pessimistic RBML technique, and Figure 15b shows the execution of the synthetic application using the HM_{PL}. If the computation is enough to cover the transmission of the received message to its protector, then the overhead in the execution time should be very low.

To finalize the experimental evaluation, we have designed a benchmark application which focuses on replying to the scenario shown in Figure 15. This is the best case scenario for the HM_{PL} and it can be observed in Figure 16, where the HM_{PL} is able to reduce the added overhead of the RBML up to fourteen times. In order to achieve such a reduction, receiver processes should execute computation that allows a total overlap of the logging operation.

Scientific parallel applications are normally composed of a set of phases that are repeated during application execution [26]. Some of these phases could be computation bound and others communication bound. If we use the temporary buffers of the HM_{PL} to save messages during communication bound phases without retransmitting them to the protectors, we can decrease overheads in these phases. Then, in computation bound phases, messages in these temporary buffers could be sent to the logger threads. This method will allow to reduce impacts in communications, therefore reducing overheads.

In order to apply the above-mentioned method, it will be necessary to firstly analyze and determine application phases. If a tool such as PAS2P is used, we will be able to determine the behavior (communication bound, computation bound, balanced) of each phase and this information could be

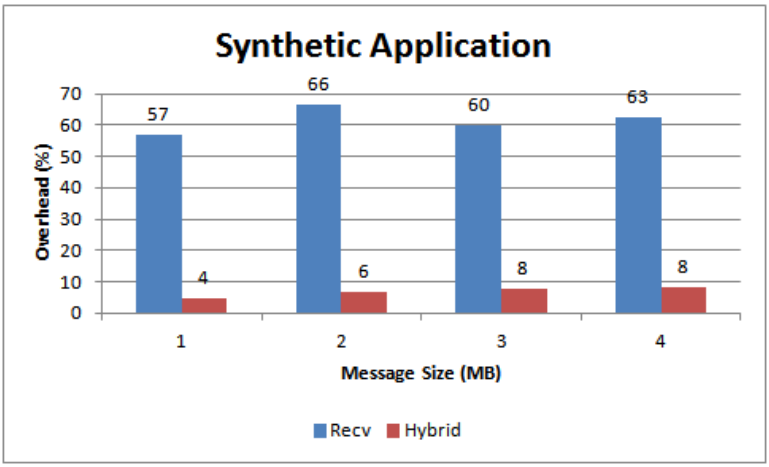


Figure 16: Overhead Analysis with the Synthetic Application.

used by the underlying fault tolerance support to determine the best moment to log messages in the logger threads.

Such a methodology is beyond the scope of this paper, and is thus left as an open line of research.

5.5. Discussion

The experiments and validation presented above help us to demonstrate that the HM_{PL} is able to outperform the RBML in failure-free and faulty scenarios. Besides that, subsection 5.4 highlights the potential of our approach when a total overlapping between computational phases and logging phases is achieved.

In this section we discuss the key aspects that may affect scalability and sensitivity of the HM_{PL}. We aim to analyze conceptually and experimentally the benefits of our approach and how it can be extended to other applications and systems.

The proposed HM_{PL} has been integrated inside the RADIC fault tolerance architecture. One of the main advantages of RADIC is that all decisions and tasks are distributed. Then, there is not a single element or task that depends on the number of nodes that are being used to execute applications [9] [16]. Moreover, RADIC acts as a distributed and decentralized controller, and global communications such as broadcasts are not used. Taking into account the HM_{PL} design (Section 4), it fulfills RADIC's characteristics as well.

Then, the scalability of the HM_{PL} is fully dependent on how the underlying system scales with the increase in the number of nodes used. If the application (in conjunction with the underlying system) does not scale properly, then RADIC and the HM_{PL} will not scale because of this. If we analyze the resource utilization of the HM_{PL} we should take into account the next:

- **Communications and Network.** Network messages are duplicated (as any other receiver-based logging approach). However, the HM_{PL} removes this operation from the critical path, and thanks to RADIC's features, these communications can be made between neighbor nodes, without needing centralized storage. Let us consider the following scenario when RADIC is being used. RADIC (using the HM_{PL}) is being executed using N nodes with a bandwidth of B between protectors and protected nodes and then we increase the number of nodes to $100 \times N$, but the bandwidth between protectors and protected nodes is still B, then RADIC will not affect the scalability, since this increment does not affect the resources that RADIC is currently using. However, if the bandwidth is reduced, then RADIC and the application will not be able to scale.

- **Memory and Storage.** The tables and data structures that RADIC and the HM_{PL} use can increase a little with the number of nodes, but as these are simple data structures with only integer values inside them, the tables can easily be allocated in memory. The RADICTable (which is used by the HM_{PL}) has one entry by application process and each row contains: process id (4 bytes), URI (256 bytes), the message Receive Sequence Number (RSN) (4 bytes) and Send Sequence Number (SSN) (4 bytes). It can be seen that with 10000 processes the size of the RADICTable that each process has to maintain in memory would be around 2.5 MB. The size of the Temporary Buffers will depend exclusively on the time spent in logging messages onto stable storage, but as mentioned before, messages in these buffers are continuously deleted once they are fully logged. The size of Temporary Buffers can also be limited, and once the limit is reached, the HM_{PL} can act as a classic Receiver Based Message Logging (RBML) with the following messages (until temporary messages are removed from the queue), in order to avoid increasing the memory usage.

- **Computation.** The HM_{PL} consumes computational resources as any

other pessimistic receiver-based approach.

In order to check the sensitivity of our approach when using high speed networks such as Infiniband, we conducted a set of basic experiments using an Infiniband environment. We created a synthetic application that pings data from node A to node B, and node C is in charge of logging messages received by node B (based on the Netpipe Tool) using packets of 400000 bytes. We executed the application in the using the next infrastructures:

- Dell PowerEdge M600 with 8 nodes, each node with 2 quad-core Intel® Xeon® E5430 running at 2.66 GHz. Each node has 16 GB of main memory and a dual embedded Broadcom® NetXtreme IITM 5708 Gigabit Ethernet. RADIC features and the message logging techniques have been integrated into Open MPI 1.7 with BLCR checkpoint library support.

- Dell PowerEdge C6145 with 4 nodes, each node with 4 processors AMD Opteron® 6200 with Mellanox ConnectX3 MC353A-QCBT (4x10Gbps) network cards.RADIC features and the message logging techniques have been integrated into Open MPI 1.7.

Table 1 shows how the performance and benefits of our proposal does not depend on the underlying network infrastructure. As it can be seen, the HM_{PL} presents an overhead reduction of around 30% when compared to the RBML in both environments.

Table 1: Hybrid Message Pessimistic Logging comparison using Infiniband and Gigabit Ethernet.

Networks	**Overhead Receiver-based (%)**	**Overhead Hybrid Pessimistic Logging (%)**	**Overhead Reduction (%)**
Gigabit Ethernet	81.42	51.07	30.35
Infiniband	85.04	52.09	32.95

6. Conclusions and Future Work

In this paper we have presented the Hybrid Message Pessimistic Logging (HM_{PL}). This technique aims to reduce the impact of pessimistic receiver-based message logging during failure-free executions without harming the recovery phase of this protocol.

The main objective of this research is to reduce the negative impact of message logging techniques in parallel executions. In order to comply with this objective, we have proposed the HM_{PL}. This is a novel pessimistic message logging approach that combines the advantages of two of the most classical message logging approaches: Sender-based Message Logging and Receiver-based Message Logging. The HM_{PL} approach focuses on providing a fault tolerant solution with low MTTR of processes by lowering the complexity of the recovery process of Sender-based approaches and, at the same time, reducing the impact of failure-free executions in comparison with receiver-based approaches.

The HM_{PL} relies on the usage of data structures to save messages temporarily (in senders and receivers) and allowing the application to continue its execution without restricting message emissions while other messages are being saved in stable storage. The results obtained have demonstrated that the HM_{PL} is able to reduce up to 34% of overhead during failure-free executions and 20% in faulty executions when comparing it with a pessimistic receiver-based message logging. However, we also have shown that the HM_{PL} has the potential to reduce the added overhead of the RBML up to fourteen times when the computation time allows the total overlap of the logging operation. In order to achieve such a reduction, receiver processes should execute computation that allows a total overlap of the logging operation.

Future work will focus on extending the analysis of the HM_{PL} to a wider set of parallel scientific applications. We will also focus our efforts on extracting application data that could be helpful in determining the best moments to log messages during application executions. By using the HM_{PL} we can save messages in the temporary buffers during communication bound stages of applications and allow the logging operation to continue during computation bound stages.

Acknowledgment

This research has been supported by the MINECO (MICINN) Spain under contracts TIN2011-24384 and TIN2014-53172-P.

References

[1] T. J. Hacker, F. Romero, C. D. Carothers, An analysis of clustered failures on large supercomputing systems, Journal of Parallel and Distributed Computing 69 (7) (2009) 652 – 665. doi:http://dx.doi.org/10.1016/j.jpdc.2009.03.007.
URL http://www.sciencedirect.com/science/article/pii/S0743731509000446

[2] B. Schroeder, G. Gibson, A large-scale study of failures in high-performance computing systems, Dependable and Secure Computing, IEEE Transactions on 7 (4) (2010) 337–350. doi:10.1109/TDSC.2009.4.

[3] I. Egwutuoha, D. Levy, B. Selic, S. Chen, A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems, The Journal of Supercomputing 65 (3) (2013) 1302–1326. doi:10.1007/s11227-013-0884-0.
URL http://dx.doi.org/10.1007/s11227-013-0884-0

[4] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surv. 34 (3) (2002) 375–408. doi:10.1145/568522.568525.
URL http://doi.acm.org/10.1145/568522.568525

[5] A. Bouteiller, G. Bosilca, J. Dongarra, Retrospect: Deterministic replay of mpi applications for interactive distributed debugging, Recent Advances in Parallel Virtual Machine and Message Passing Interface 4757 (2007) 297–306. doi:10.1007/978-3-540-75416-9_41.
URL http://dx.doi.org/10.1007/978-3-540-75416-9_41

[6] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565. doi:10.1145/359545.359563.
URL http://doi.acm.org/10.1145/359545.359563

[7] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, F. Cappello, Improved Message Logging Versus Improved Coordinated Checkpointing for Fault Tolerant MPI, in: Cluster Computing, 2004 IEEE International Conference on, Vol. 0, IEEE Computer Society, Los Alamitos, CA, USA, 2004, pp. 115–124. doi:http://doi.ieeecomputersociety.org/10.1109/CLUSTR.2004.1392609.

[8] H. Meyer, D. Rexachs, E. Luque, Hybrid Message Logging. Combining advantages of Sender-based and Receiver-based Approaches, Procedia Computer Science 29 (0) (2014) 2380 – 2390, 2014 International Conference on Computational Science. doi:http://dx.doi.org/10.1016/j.procs.2014.05.222.
URL http://www.sciencedirect.com/science/article/pii/S1877050914003998

[9] A. Duarte, D. Rexachs, E. Luque, Increasing the cluster availability using radic, in: Cluster Computing, 2006 IEEE International Conference on, 2006, pp. 1–8. doi:10.1109/CLUSTR.2006.311872.

[10] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, D. Arnold, Evaluating the viability of process replication reliability for exascale systems, Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011) 44:1–44:12doi:10.1145/2063384.2063443.
URL http://doi.acm.org/10.1145/2063384.2063443

[11] J. Xu, R. Netzer, M. Mackey, Sender-based message logging for reducing rollback propagation, Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on (1995) 602–609doi:10.1109/SPDP.1995.530738.

[12] S. Rao, L. Alvisi, H. Vin, The cost of recovery in message logging protocols, Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on (1998) 10–18doi:10.1109/RELDIS.1998.740469.

[13] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, F. Cappello, MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI., IJHPCA.

[14] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, J. Dongarra, Reasons for a Pessimistic or Optimistic Message Logging Protocol in MPI Uncoordinated Failure Recovery, in: IEEE International Conference on Cluster Computing (Cluster 2009), New Orleans, États-Unis, 2009, pp. 229–236.

[15] A. Bouteiller, G. Bosilca, J. Dongarra, Redesigning the message logging model for high performance, Concurr. Comput. : Pract. Exper. (2010) 2196–2211.

[16] G. Santos, L. Fialho, D. Rexachs, E. Luque, Increasing the availability provided by RADIC with low overhead, in: Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on, 2009, pp. 1–8. doi:10.1109/CLUSTR.2009.5289163.

[17] J. C. Y. Ho, C.-L. Wang, F. C. M. Lau, Scalable group-based checkpoint/restart for large-scale message-passing systems, in: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, 2008, pp. 1–12.

[18] Y. Luo, D. Manivannan, Hope: A hybrid optimistic checkpointing and selective pessimistic message logging protocol for large scale distributed systems, Future Generation Computer Systems 28 (8) (2012) 1217–1235.

[19] T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. V. Kalé, F. Cappello, On the use of cluster-based partial message logging to improve fault tolerance for MPI HPC applications, in: Proceedings of the 17th International Conference on Parallel Processing - Volume Part I, Euro-Par'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 567–578.

[20] M. Castro, D. Rexachs, E. Luque, Adding semi-coordinated checkpoint to radic in multicore clusters, in: PDPTA 2013, 2013, pp. 545–551.

[21] A. Bouteiller, T. Herault, G. Bosilca, J. J. Dongarra, Correlated set coordination in fault tolerant message logging protocols for many-core clusters, Concurrency and Computation: Practice and Experience 25 (4) (2013) 572–585. doi:10.1002/cpe.2859.
URL `http://dx.doi.org/10.1002/cpe.2859`

[22] D. B. Johnson, W. Zwaenepoel, Sender-based message logging, In Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing (1987) 14–19.

[23] H. Meyer, D. Rexachs, E. Luque, RADIC: A fault tolerant middleware with automatic management of spare nodes, The 2012 International Conference on Parallel and Distributed Processing Techniques and Applications, July 16-19, Las Vegas, USA (2012) 17–23.

[24] Q. O. Snell, A. R. Mikler, J. L. Gustafson, NetPIPE: A Network Protocol Independent Performance Evaluator, In IASTED International Conference on Intelligent Information Management and Systems.

[25] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, The Nas Parallel Benchmarks, International Journal of High Performance Computing Applications.

[26] A. Wong, D. Rexachs, E. Luque, Parallel application signature for performance analysis and prediction (TPDS) (accepted, available online), IEEE Transactions on Parallel and Distributed Systems 99. doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2329688.